Building a Weather App

We will be building off of the sample application to create a weather app (you can create a new one for this example with `react-native init WeatherProject`). This will give us a chance to explore how to utilize and combine stylesheets, flexbox, network communication, user input, and images into a useful app we can then deploy to an Android or iOS device.

This section may feel like a bit of a blur, as we'll be focusing on an overview of these features rather than deep explanations of them. The Weather App will serve as a useful reference in future sections as we discuss these features in more detail. Don't worry if it feels like we're moving quickly!

As shown in Figure 3-13, the final application includes a text field where users can input a zip code. It will then fetch data from the OpenWeatherMap API and display the current weather.

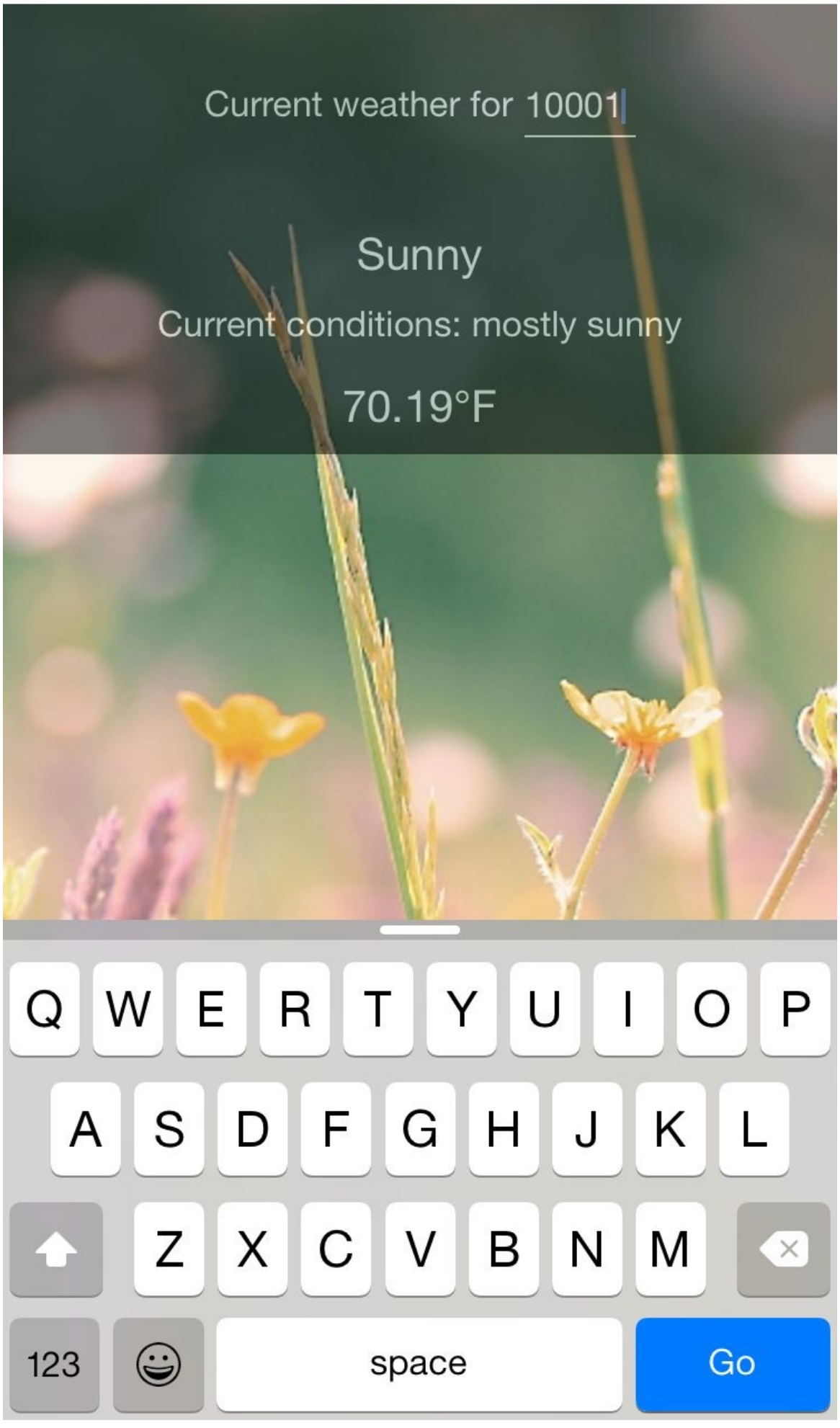
Carrier
11:17 PM
Current weather for 10001
Sunny
Current conditions: mostly sunny
70.19°F
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
123
space
Go

Figure 3-13. The finished weather app

The first thing we'll do is replace the default code. Move the initial component out into its own file, *WeatherProject.js*, and replace the contents of *index.ios.js* and *index.android.js*.

Example 3-6. Simplified contents of index.ios.js and index.android.js (they should be identical)

```
var React = require('react-native');
var { AppRegistry } = React;
var WeatherProject = require('./WeatherProject');
AppRegistry.registerComponent('WeatherProject', () => WeatherProject);
```

Handling User Input

We want the user to be able to input a zip code and get the forecast for that area, so we need to add a text field for user input. We can start by adding zip code information to our component's initial state (see Example 3-7).

Example 3-7. Add this to your component, before the render function

```
getInitialState: function() {
  return {
    zip: ''
  };
}
```

Remember that `getInitialState` is how we set up the initial `state` values for React components. If you need a review of the React component lifecycle, see the React docs.

Then, we should also change one of the `<Text>` components to display `this.state.zip`:

```
<Text style={styles.welcome}>
  You input {this.state.zip}.
</Text>
```

With that out of the way, let's add a `<TextInput>` component (this is a basic component that allows the user to enter text):

```
<TextInput
  style={styles.input}
  onSubmitEditing={this._handleTextChange}/>
```

The `<TextInput>` component is documented on the React Native site, along with its properties. You can also pass the `<TextInput>` additional callbacks in order to listen to other events, such as `onChange` or `onFocus`, but we do not need them at the moment.

Note that we've added a simple style to the `<TextInput>`. Add the `input` style to your stylesheet:

```
var styles = StyleSheet.create({
  ...
  input: {
    fontSize: 20,
    borderWidth: 2,
    height: 40
    }
  ...
});
```

The callback we passed as the `onSubmitEditing` prop looks like this, and should be added as a function on the component:

```
_handleTextChange(event) {
 console.log(event.nativeEvent.text);
 this.setState({zip: event.nativeEvent.text})
}
```

The console statement is extraneous, but it will allow you to test out the debugger tools if

you so desire.

You will also need to update your import statements:

```
var React = require('react-native');
var {
  ...
  TextInput
  ...
} = React;
```

Now, try running your application using the iOS simulator. It won't be pretty, but you should be able to successfully submit a zip code and have it be reflected in the `<Text>` component.

If we wanted, we could add some simple input validation here to ensure that the user typed in a five-digit number, but we will skip that for now.

Example 3-8 shows the full code for the *WeatherProject.js* component.

Example 3-8. WeatherProject.js: this version simply accepts and records user input

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TextInput,
  Image
} = React;


var WeatherProject = React.createClass({
  // If you want to have a default zip code, you could add one here
  getInitialState() {
    return ({
      zip: ''
    });
  },
  // We'll pass this callback to the <TextInput>
  _handleTextChange(event) {

    // log statements are viewable in Xcode,
    // or the Chrome debug tools
    console.log(event.nativeEvent.text);

    this.setState({
      zip: event.nativeEvent.text
    });
  },
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          You input {this.state.zip}.
        </Text>
        <TextInput
              style={styles.input}
              onSubmitEditing={this._handleTextChange}/>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
```

```
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  input: {
    fontSize: 20,
    borderWidth: 2,
    height: 40
    }
});

module.exports = WeatherProject;
```

Displaying Data

Now let's work on displaying the forecast for that zip code. We will start by adding some mock data to `getInitialState` in *WeatherProject.js*:

```
  getInitialState() {
    return {
      zip: '',
      forecast: {
        main: 'Clouds',
        description: 'few clouds',
        temp: 45.7
      }
    }
  }
```

For sanity's sake, let's also pull the forecast rendering into its own component. Make a new file called *Forecast.js* (see Example 3-9).

Example 3-9. Forecast component in Forecast.js

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View
} = React;

var Forecast = React.createClass({
  render: function() {
    return (
      <View>
        <Text style={styles.bigText}>
          {this.props.main}
        </Text>
        <Text style={styles.mainText}>
          Current conditions: {this.props.description}
        </Text>
        <Text style={styles.bigText}>
          {this.props.temp}°F
        </Text>
      </View>
    );
  }
});


var styles = StyleSheet.create({
  bigText: {
    flex: 2,
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
    color: '#FFFFFF'
  },
  mainText: {
    flex: 1,
    fontSize: 16,
    textAlign: 'center',
    color: '#FFFFFF'
  }
})

module.exports = Forecast;
```

The `<Forecast>` component just renders some `<Text>` based on its props. We've also included some simple styles at the bottom of the file, to control things like text color.

Require the `<Forecast>` component and then add it to your app's render method, passing

it props based on the `this.state.forecast` (see Example 3-10). We'll address issues with layout and styling later. You can see how the `<Forecast>` component appears in the resulting application in Figure 3-14.

Example 3-10. WeatherProject.js should be updated with new state and the Forecast component

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TextInput,
  Image
} = React;

var Forecast = require('./Forecast');

var WeatherProject = React.createClass({
  getInitialState() {
    return {
      zip: '',
      forecast: {
        main: 'Clouds',
        description: 'few clouds',
        temp: 45.7
      }
    }
  },
  _handleTextChange(event) {
    console.log(event.nativeEvent.text);
    this.setState({
      zip: event.nativeEvent.text
    });
  },
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          You input {this.state.zip}.
        </Text>
        <Forecast
          main={this.state.forecast.main}
          description={this.state.forecast.description}
          temp={this.state.forecast.temp}/>
        <TextInput
          style={styles.input}
          returnKeyType='go'
          onSubmitEditing={this._handleTextChange}/>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#4D4D4D',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  input: {
    fontSize: 20,
    borderWidth: 2,
    height: 40
    }
});
```

```
module.exports = WeatherProject;
```

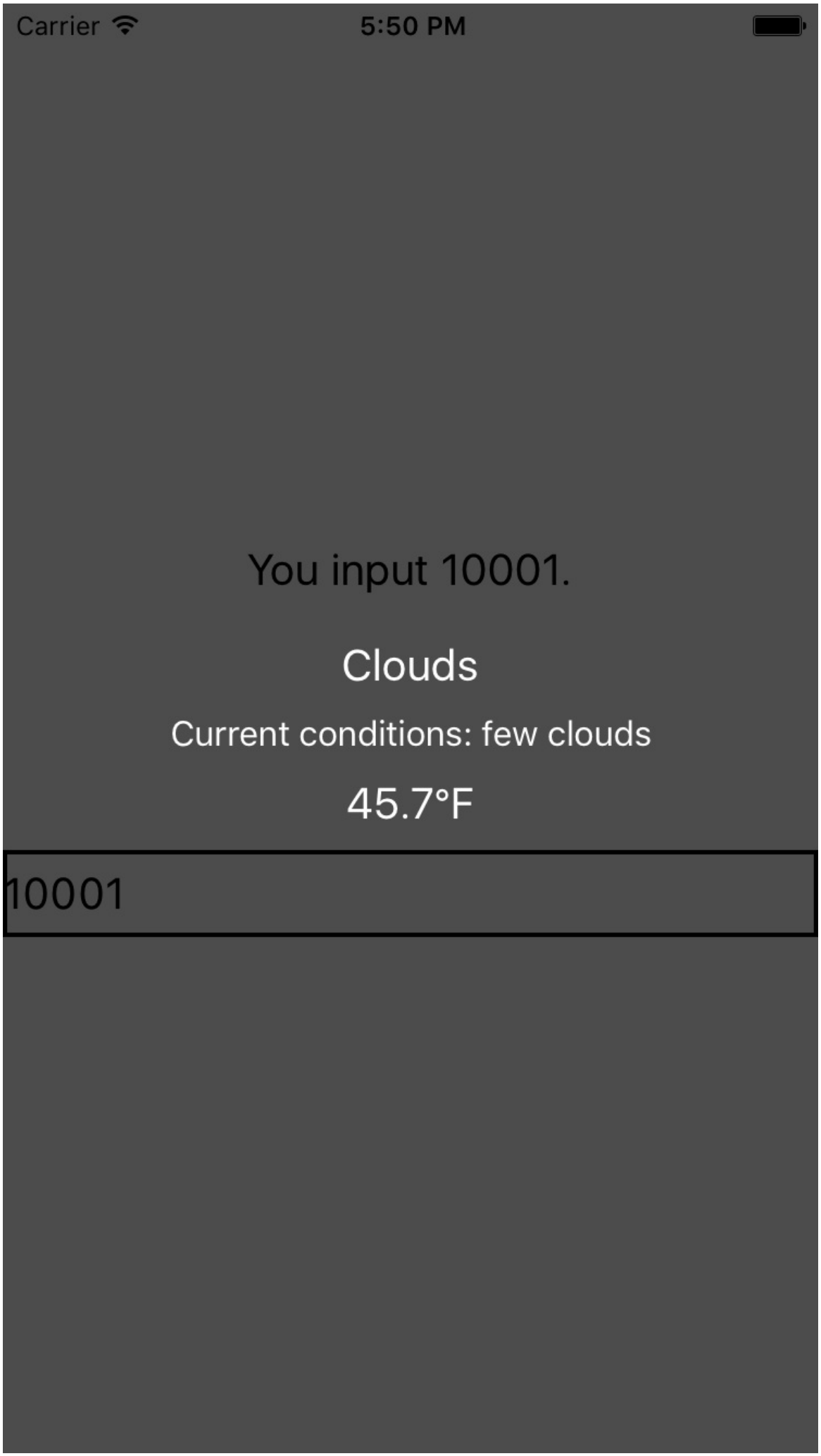
Carrier
5:50 PM
You input 10001.
Clouds
Current conditions: few clouds
45.7°F
10001

Figure 3-14. The weather app so far

Adding a Background Image

Plain background colors are boring. Let’s display a background image to go along with our forecast.

ASSET INCLUSION IS PLATFORM-SPECIFIC

Android and iOS have different requirements for adding assets to your projects. We'll cover both here.

Assets such as images need to be added to your project based on which platform you're building for. We'll start with Xcode.

Select the *Images.xcassets/* folder, and then select the New Image Set option, as shown in Figure 3-15. Then, you can drag and drop an image into the set. Figure 3-16 shows the resulting Image Set. Make sure the image set's name matches the filename, otherwise React Native will have difficulty importing it.

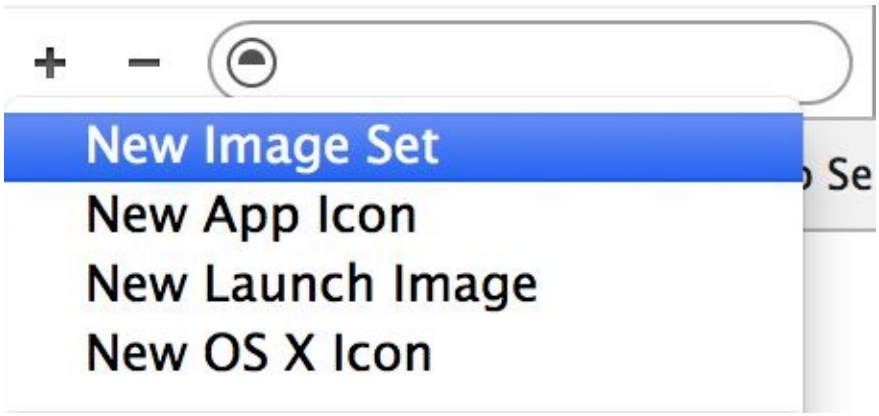

Figure 3-15. Add a new image set

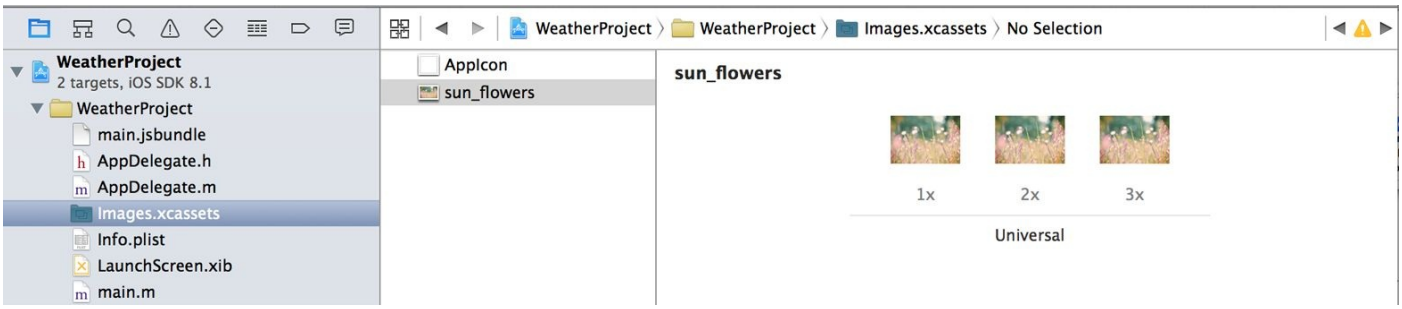

Figure 3-16. Drag your image files into the image set to add them

The @2x and @3x decorators indicate an image with a resolution of twice and thrice the base resolution, respectively. Because the WeatherApp is designated as a universal application (meaning one that can run on iPhone or iPad), Xcode gives us the option of uploading images at the various appropriate resolutions.

For Android, we have to add our files as bitmap drawable resources to the appropriate folders in *WeatherProject/android/app/src/main/res*. You'll want to copy the *.png* file into the following resolution-specific directories (see Figure 3-17):

- *drawable-mdpi/* (1x)
- *drawable-hdpi/* (1.5x)
- *drawable-xhdpi/* (2x)
- *drawable-xxhdpi/* (3x)

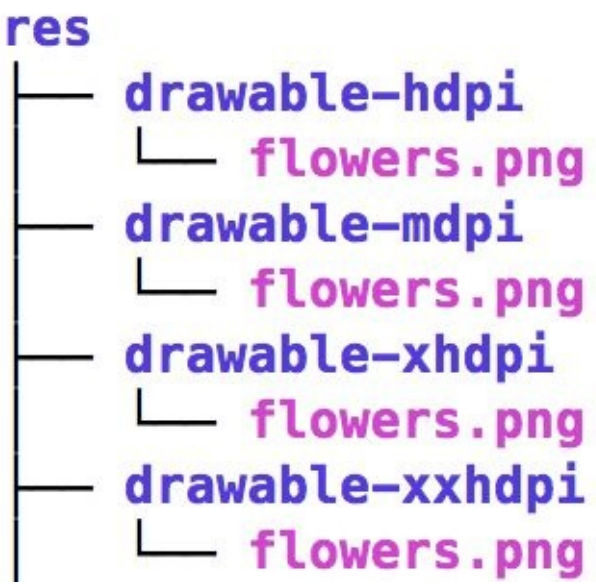

Figure 3-17. Adding image files to Android

After that, the image will be available to your Android application.

If this workflow feels suboptimal, that's because it is. It will probably change in future versions of React Native.

Now that the image files have been imported into both our Android and iOS projects, let's hop back to our React code. To add a background image, we don't set a `background` property on a `<div>` like we can do on the Web. Instead, we use an `<Image>` component as a container:

```
<Image source={require('image!flowers')}
       resizeMode='cover'
       style={styles.backdrop}>
  // Your content here
</Image>
```

The `<Image>` component expects a `source` prop, which we get by using `require`. The call to `require(image!flowers)` will cause React Native to search for a file named *flowers*.

Don't forget to style it with `flexDirection` so that its children render as we'd like them to:

```
backdrop: {
  flex: 1,
  flexDirection: 'column'
}
```

Now let's give the `<Image>` some children. Update the `render` method of the `<WeatherProject>` component to return the following:

```
<Image source={require('image!flowers')}
       resizeMode='cover'
       style={styles.backdrop}>
  <View style={styles.overlay}>
   <View style={styles.row}>
     <Text style={styles.mainText}>
```

```
        Current weather for
      </Text>
      <View style={styles.zipContainer}>
        <TextInput
          style={[styles.zipCode, styles.mainText]}
          returnKeyType='go'
          onSubmitEditing={this._handleTextChange}/>
      </View>
    </View>
    <Forecast
       main={this.state.forecast.main}
       description={this.state.forecast.description}
       temp={this.state.forecast.temp}/>
  </View>
</Image>
```

You'll notice that I'm using some additional styles that we haven't discussed yet, such as `row`, `overlay`, and the `zipContainer` and `zipCode` styles. You can skip ahead to the end of this section to see the full stylesheet.

Fetching Data from the Web

Next, let's explore using the networking APIs available in React Native. You won't be using jQuery to send AJAX requests from mobile devices! Instead, React Native implements the Fetch API. The Promise-based syntax is fairly simple:

```
fetch('http://www.somesite.com')
  .then((response) => response.text())
  .then((responseText) => {
    console.log(responseText);
  });
```

We will be using the OpenWeatherMap API, which provides us with a simple endpoint that returns the current weather for a given zip code.

To integrate this API, we can change the callback on the `<TextInput>` component to query the OpenWeatherMap API:

```
_handleTextChange: function(event) {
  var zip = event.nativeEvent.text;
  this.setState({zip: zip});
  fetch('http://api.openweathermap.org/data/2.5/weather?q=' +
  zip + '&units=imperial')
    .then((response) => response.json())
    .then((responseJSON) => {
      // Take a look at the format, if you want.
      console.log(responseJSON);
      this.setState({
        forecast: {
          main: responseJSON.weather[0].main,
          description: responseJSON.weather[0].description,
          temp: responseJSON.main.temp
        }
      });
    })
    .catch((error) => {
      console.warn(error);
    });
}
```

Note that we want the JSON from the response. The Fetch API is pretty straightforward to work with, so this is all we will need to do.

The other thing that we can do is to remove the placeholder data, and make sure that the forecast does not render if we do not have data yet.

First, clear the mock data from `getInitialState`:

```
getInitialState: function() {
  return {
    zip: '',
    forecast: null
  };
}
```

Then, in the render function, update the rendering logic:

```
var content = null;
if (this.state.forecast !== null) {
  content = <Forecast
```

```
                main={this.state.forecast.main}
                description={this.state.forecast.description}
                temp={this.state.forecast.temp}/>;
  }
```

Finally, replace your rendered `<Forecast>` component with `{content}` in the `render` function.

Putting It Together

For the final version of the application, I've reorganized the `<WeatherProject>` component's `render` function and tweaked the styles. The main change is to the layout logic, diagrammed in Figure 3-18.

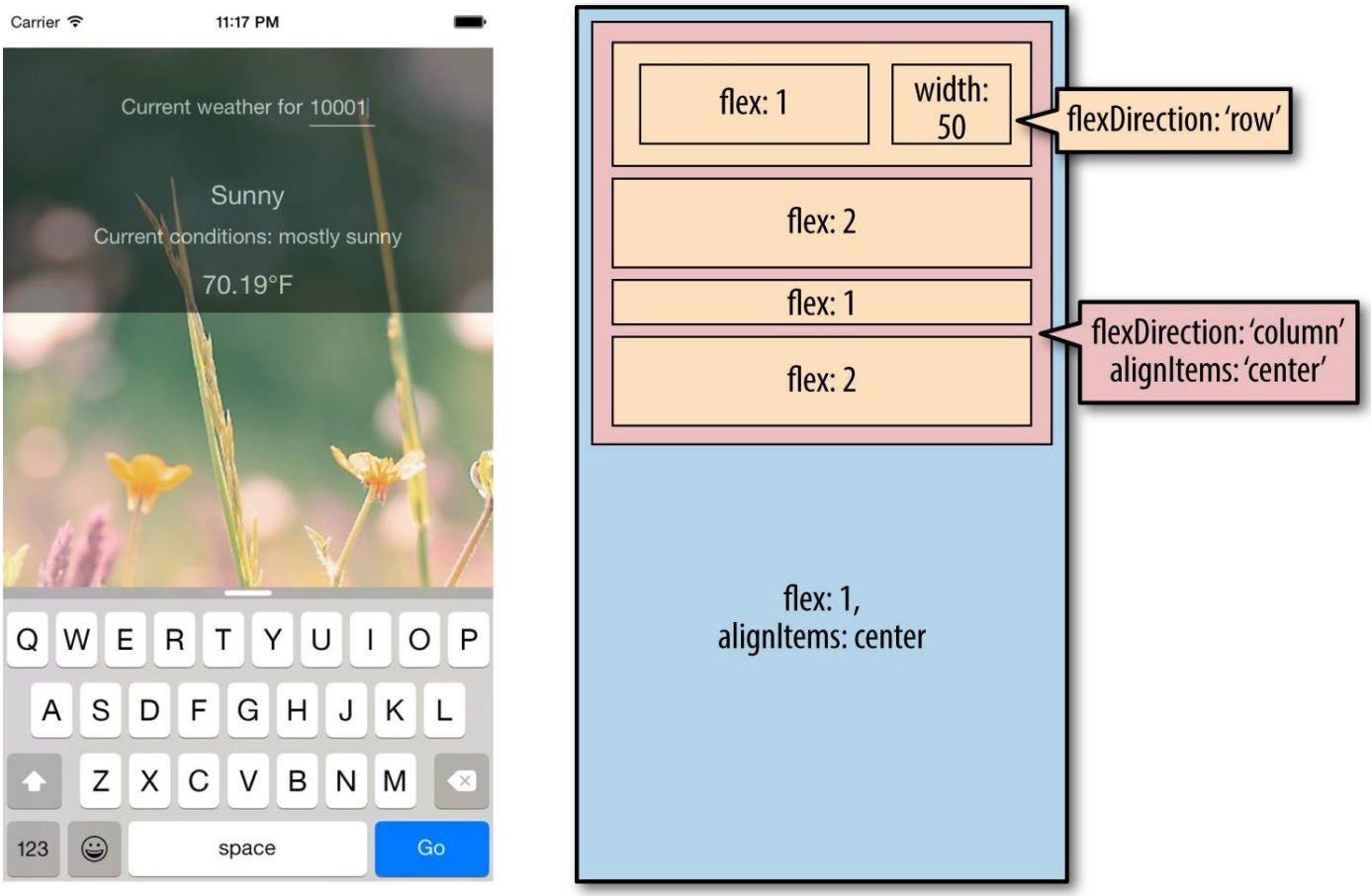

Figure 3-18. Layout of the finished weather application

OK. Ready to see it all in one place? Example 3-11 shows the finished code for the `<WeatherProject>` component in full, including the stylesheets. The `<Forecast>` component will be the same as above in Example 3-9.

Example 3-11. Finished code for WeatherProject.js

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TextInput,
  Image
} = React;
var Forecast = require('./Forecast');

var WeatherProject = React.createClass({
  getInitialState: function() {
    return {
      zip: '',
      forecast: null
    };
  },

  _handleTextChange: function(event) {
    var zip = event.nativeEvent.text;
    this.setState({zip: zip});
```

```
    fetch('http://api.openweathermap.org/data/2.5/weather?q='
      + zip + '&units=imperial')
      .then((response) => response.json())
      .then((responseJSON) => {
        this.setState({
          forecast: {
            main: responseJSON.weather[0].main,
            description: responseJSON.weather[0].description,
            temp: responseJSON.main.temp
          }
        });
      })
      .catch((error) => {
        console.warn(error);
      });
  },

  render: function() {
    var content = null;
    if (this.state.forecast !== null) {
      content = <Forecast
                  main={this.state.forecast.main}
                  description={this.state.forecast.description}
                  temp={this.state.forecast.temp}/>;
    }
    return (
      <View style={styles.container}>
        <Image source={require('image!flowers')}
               resizeMode='cover'
               style={styles.backdrop}>
          <View style={styles.overlay}>
           <View style={styles.row}>
             <Text style={styles.mainText}>
               Current weather for
             </Text>
             <View style={styles.zipContainer}>
               <TextInput
                 style={[styles.zipCode, styles.mainText]}
                 returnKeyType='go'
                 onSubmitEditing={this._handleTextChange}/>
             </View>
           </View>
           {content}
         </View>
        </Image>
      </View>
    );
  }
});

var baseFontSize = 16;

var styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    paddingTop: 30
  },
  backdrop: {
    flex: 1,
    flexDirection: 'column'
  },
  overlay: {
    paddingTop: 5,
    backgroundColor: '#000000',
    opacity: 0.5,
    flexDirection: 'column',
    alignItems: 'center'
  },
  row: {
    flex: 1,
    flexDirection: 'row',
    flexWrap: 'nowrap',
    alignItems: 'flex-start',
    padding: 30
  },
```

```
  zipContainer: {
    flex: 1,
    borderBottomColor: '#DDDDDD',
    borderBottomWidth: 1,
    marginLeft: 5,
    marginTop: 3
  },
  zipCode: {
    width: 50,
    height: baseFontSize,
  },
  mainText: {
    flex: 1,
    fontSize: baseFontSize,
    color: '#FFFFFF'
  }
});

module.exports = WeatherProject;
```

Now that we’re done, try launching the application. It should work on both Android and iOS, in an emulator or on your physical device. What would you like to change or improve?

You can view the completed application in the GitHub repository.

Summary

For our first real application, we've already covered a lot of ground. We introduced a new UI component, `<TextInput>`, and learned how to use it to get information from the user. We demonstrated how to implement basic styling in React Native, as well as how to use images and include assets in our application. Finally, we learned how to use the React Native networking API to request data from external web sources. Not bad for a first application!

Hopefully, this has demonstrated how quickly you can build React Native applications with useful features that feel at home on a mobile device.

If you want to extend your application further, here are some things to try:

- Add more images, and change them based on the forecast
- Add validation to the zip code field
- Switch to using a more appropriate keypad for the zip code input
- Display the five-day weather forecast

Once we cover more topics, such as geolocation, you will be able to extend the weather application in even more ways.

Of course, this has been a pretty quick survey. In the next few chapters, we will focus on gaining a deeper understanding of React Native best practices, and look at how to use a lot more features, too!